Lock-free/Wait-free

For non-blocking concurrent operations

Non-blocking

- Surprisingly lock-free and wait-free do not mean your application does not lock or wait. Here are the definitions.
- From Wikipedia
- In computer science, an algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread; for some operations, these algorithms provide a useful alternative to traditional blocking implementations. A non-blocking algorithm is lock-free if there is guaranteed system-wide progress, and wait-free if there is also guaranteed per-thread progress.

Why Non-blocking

- Avoid issues in standard multi-threading/processes
- Such as mutexes, semaphores etc.
- Deadlocks
- Live-locks
- Priority Inversions
- Resource starvation due to many processes locking the same resources.

Tools for Non-Blocking

- Atomic Operations
- Compare and Swap (CAS)
- Fetch and Add (FFA)
- Future discussions, Read-Copy-Update and the Linux Kernel

Atomic Operations

- To accomplish non-blocking there has to be some “atomic” operations that are guaranteed to perform without corruption or interference from competing threads or processes.

- Some algorithms require “atomic” to be applied to the MMA controller as well.

Compare and Swap

- Wikipedia pseudo code:

```
function cas(p : pointer to int, old : int, new : int) returns bool
{
    if *p ≠ old {
        return false
    }
    *p ← new
    return true
}
```

ABA Problem

- CAS has one issue to deal with. Again from Wikipedia:

Some CAS-based algorithms are affected by and must handle the problem of a false positive match, or the ABA problem. It's possible that between the time the old value is read and the time CAS is attempted, some other processors or threads change the memory location two or more times such that it acquires a bit pattern which matches the old value. The problem arises if this new bit pattern, which looks exactly like the old value, has a different meaning: for instance, it could be a recycled address, or a wrapped version counter.

There are simple ways to deal with this, but are scenario dependent.

Concurrent Linked List

- Paper illustrating:
- http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=1FC420B6D62DF1F27CD78AFCF9F8CB78?doi=10.1.1.41.9506&rep=rep1&type=pdf

- Uses CAS

- The paper is a pdf that prevents me from copy and pasting text, so I am either taking screenshots or paraphrasing.

- This is involved gentlemen, please bear with me.
- To implement a lock free linked list, access is trivial, but deletions cause mucho problemos.
- If process a is deleting a cell, and process b is deleting an adjacent cell, the integrity of the list is at risk. So the paper describes the solution as follows:
- The linked list is made up of normal cells which have a next field and all content fields.
- The linked list also has auxiliary cells which consist only of the next field.
- Each normal cell must be preceded by and succeeded by auxiliary cells.
- There can be any number of auxiliary cells between normal cells but there must be at least one.
- This list also has 2 dummy cells, the first and last cells in the list. First points to a dummy cell and Last points to a dummy cell as well.
- The cursor is implemented by 3 pointers, pre-auxiliary cell, normal cell and post-auxiliary cell.
- By interposing auxiliary cells, multiple threads manipulating adjacent cells will not interfere with each other, assuring list integrity. CAS ensure's list integrity if more then one thread is accessing the same cell.

The structure of the cursor should be:

c.pre_aux – auxiliary cell before
c.target - is a pointer to the cell the cursor is visiting
c.pre_cell – the normal cell
c.post_aux – auxiliary cell afterwards

The paper mentions if c.pre-aux.next = c.target we have a valid cursor.

Not all aspects of this paper I fully understand. At this time I am not sure what the difference of c.target and c.pre_cell is.

```
COMPARE&SWAP(a : address, old, new : word)
  returns boolean

    BEGIN ATOMIC
1       if a^ ≠ old
2           a^ ← new
3           return TRUE
4       else
5           return FALSE
    END ATOMIC
```

Figure 1: The COMPARE&SWAP synchronization primitive.

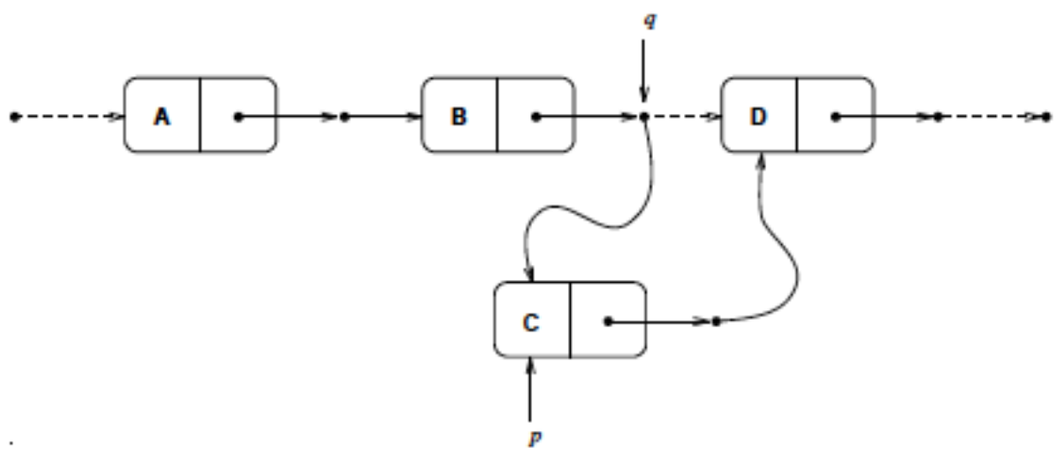

Figure 8: Inserting a new cell and auxiliary node.

```
TryDelete(c : cursor)
  returns boolean

 1      d ← c^.target
 2      n ← c^.target^.next
 3      r ← CSW(c^.pre_aux^.next, d, n)
 4      if r ≠ TRUE
 5        return FALSE
 6      Write(d^.back_link, c^.pre_cell)
 7      p ← c^.pre_cell
 8      while p^.back_link ≠ NULL
 9        q ← SafeRead(p^.back_link)
10        Release(p)
11        p ← q
12      s ← SafeRead(p^.next)
13      while n^.next^ is not a normal cell
14        q ← SafeRead(n^.next)
15        Release(n)
16        n ← q
        repeat
17        r ← CSW(p^.next, s, n)
18        if r = FALSE
19          Release(s)
20          s ← SafeRead(p^.next)
21      until r = TRUE
            or p^.back_link ≠ NULL
            or n^.next is not a normal cell
22      Release(p)
23      Release(s)
24      Release(n)
25      return TRUE
```

Figure 10: The TryDelete algorithm.

```
Delete(k : key)

 1      First(c)
 loop:
 2      r ← FindFrom(k, c)
 3      if r = FALSE
 4        return
 5      r ← TryDelete(c)
 6      if r = TRUE
 7        return
 8      Update(c)
 9      goto loop
```

Figure 13: The Delete algorithm.

I will admit I do not completely understand this slide.

According to the paper, this algorithm can be adapted for use in Binary Search Trees, and I suppose B-Tree's as well.

Fetch and Add

```
<< atomic >>
function FetchAndAdd(address location, int inc) {
    int value := *location
    *location := value + inc
    return value
}
```

http://chaoran.me/assets/pdf/wfq-ppopp16.pdf

https://github.com/chaoran/fast-wait-free-queue

A Ticket Lock/FFA Concurrent Queue

```
record locktype {
   int ticketnumber
   int turn
}
procedure LockInit( locktype* lock ) {
   lock.ticketnumber := 0
   lock.turn := 0
}
procedure Lock( locktype* lock ) {
   int myturn := FetchAndIncrement( &lock.ticketnumber )
   //must be atomic, since many threads might ask for a lock at the same time
   while lock.turn ≠ myturn
       skip // spin until lock is acquired
}
procedure UnLock( locktype* lock ) {
   FetchAndIncrement( &lock.turn )
//this need not be atomic, since only the possessor of the lock will execute this
}
```

Future Talk

- The Linux Kernel uses an atomic operation RCU
- Read-Copy-Update
- In multiple places to control concurrency with user pages and direct I/O.
- This is the work of Nick Piggin

Interesting Reading

- https://en.wikipedia.org/wiki/Fetch-and-add
- https://en.wikipedia.org/wiki/Compare-and-swap
- https://en.wikipedia.org/wiki/Ticket_lock
- https://en.wikipedia.org/wiki/Read-copy-update
- http://lwn.net/Articles/275808/
- http://lwn.net/Articles/291826/

These are here in case they come up.

Otherwise please ignore.

```
TryInsert(c : cursor, q : cell^, a : aux. node^)
  returns boolean

1       Write(q^.next, a)
2       Write(a^.next, c^.target)
3       r ← CSW(c^.pre_aux, c^.target, q)
4       return r
```

Figure 9: The TryInsert algorithm.

```
Next(c : cursor)
  returns boolean

1       if c^.target = Last
2          return FALSE
3       Release(c^.pre_cell)
4       c^.pre_cell ← SafeRead(c^.target)
5       Release(c^.pre_aux)
6       c^.pre_aux ← SafeRead(c^.target^.next)
7       Update(c)
8       return TRUE
```

Figure 7: The Next algorithm.

```
UPDATE(c : cursor)

1    if c^.pre_aux^.next = c^.target
2       return
3    p ← c^.pre_aux
4    n ← SAFEREAD(p^.next)
5    RELEASE(c^.target)
6    while n ≠ Last and n^ is not a normal cell
7       COMPARE&SWAP(c^.pre_cell^.next, p, n)
8       RELEASE(p)
9       p ← n
10      n ← SAFEREAD(p^.next)
11   c^.pre_aux ← p
12   c^.target ← n
```

Figure 5: The cursor UPDATE algorithm.

```
FIRST(c : cursor)

1    c^.pre_cell ← SAFEREAD(First)
2    c^.pre_aux ← SAFEREAD(First^.next)
3    c^.target ← NULL
4    UPDATE(c)
```

Figure 6: The FIRST algorithm.

Live Lock

Wikipedia:

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. This term was defined formally at some time during the 1970s—an early sighting in the published literature is in Babich's 1979 article on program correctness.[10] Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

Priority Inversion

Wikipedia:

In computer science, priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.

Consider two tasks H and L, of high and low priority respectively, either of which can acquire exclusive use of a shared resource R. If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes the resource. Sharing an exclusive-use resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time. Despite good design, however, it is possible that a third task M of medium priority ($p(L) < p(M) < p(H)$, where $p(x)$ represents the priority for task x) becomes runnable during L's use of R. At this point, M being higher in priority than L, preempts L, causing L to not be able to relinquish R promptly, in turn causing H—the highest priority process—to be unable to run. This is called priority inversion where a higher priority task is preempted by a lower priority one.

RCU

https://en.wikipedia.org/wiki/Read-copy-update